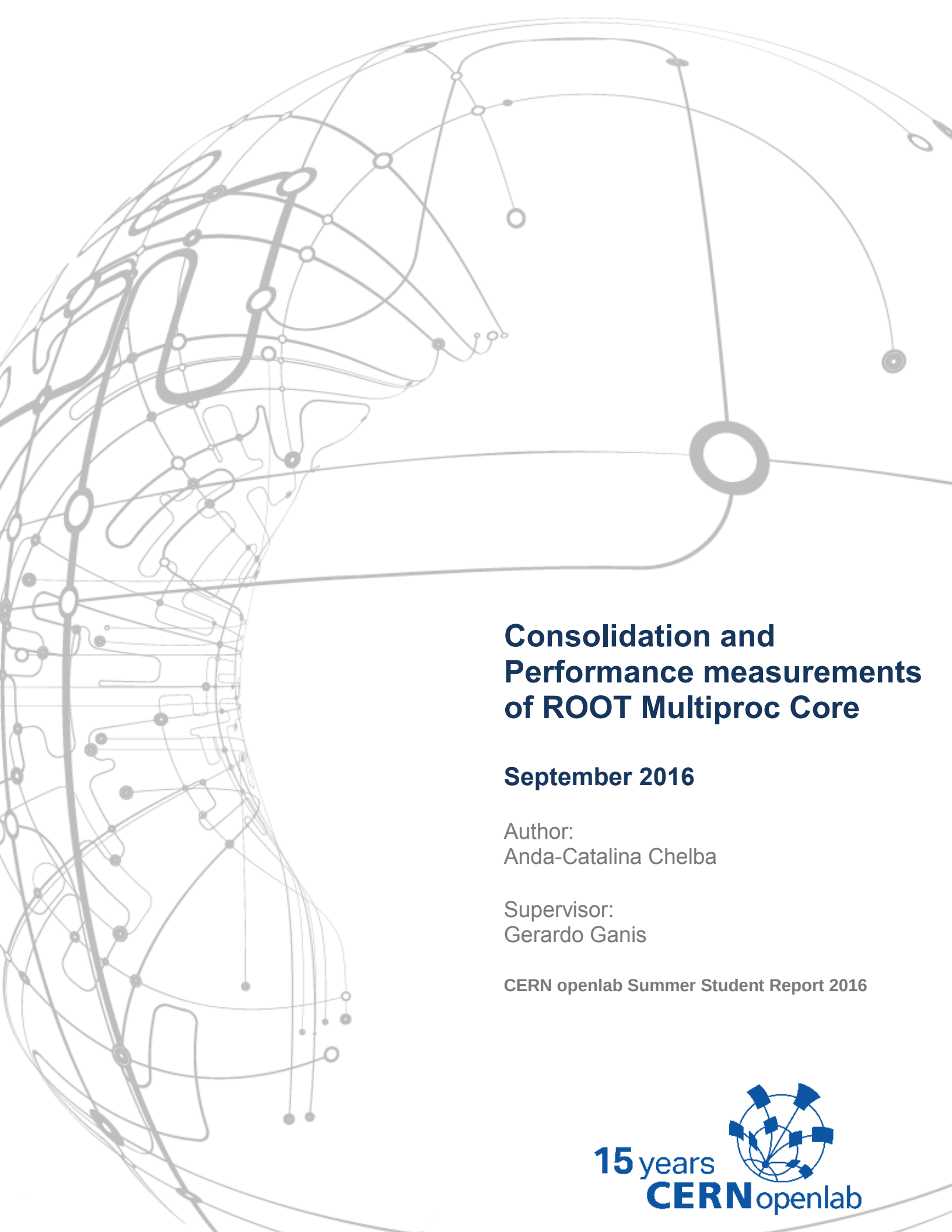

Consolidation and Performance measurements of ROOT Multiproc Core

September 2016

Author:
Anda-Catalina Chelba

Supervisor:
Gerardo Ganis

CERN openlab Summer Student Report 2016

15 years CERN openlab

Project Specification

ROOT is a C++ framework for data processing, storage and analysis born in the heart of high-energy physics research, at CERN. It is widely adopted by HEP and other scientific communities. A cornerstone of the software stacks of all LHC experiments, it is deployed for all stages of their data processing: from filtering of collision events to final analysis before publications.

ROOT features three modules allowing to process tasks in parallel: PROOF, PROOF-Lite and MultiProc. The release of this project is the consolodation of Multiproc module and the investigation of the runtime performance of different parallel processing technologies.

Acknowledgements

First and foremost, I would like to thank my supervisor, Gerardo Ganis, for his constant support provided during all phases of my work. His help and valuable inputs helped me to advance on the problems I faced. I am grateful to have been involved in such a project.

I would like to thank the CERN Openlab team for giving me the opportunity to work on this project. I would like to thank Kristina Gunne for helping me with all the administrative issues.

Lastly, I want to thank to all summer students for making the summer an unforgettable experience.

Table of Contents

1 Parallel Data Processing in ROOT

1.1 PROOF

One possible tool to run ROOT data analytics in a distributed parallel environment is PROOF. Designed as a three-tier architecture (master-client-workers), PROOF can be used in multiple scopes ranging from data-mining to long runs simulations (e.g. Monte Carlo). The master's role is to dynamically dispatch the workloads and collect computational results.

On its first versions, PROOF was designed to be a multi-node parallelism environment, thus it would benefit from the results generated by multiple machines instead of multiples cores on a single machine. This kind of architecture had its drawbacks. The nodes need to be provisioned via an initial setup phase that needs to run on each of them. This process requires a specific process that needs to be run on each node at the bootstrap phase of the node. This process is waiting for connections requests that contains as payload the work commands. Furthermore, it is pushing towards the client its status ilde.

1.2 PROOF-Lite

A version of PROOF optimized for multi-process parallelism on multi-core machines is PROOF-Lite. It evolved within the HEP community thanks to its simple method of taking advantage of multi-cores. Instead of assigning workers to different machines, the master assigns work process to different cores of the processor.

One of the biggest drawback of PROOF-Lite comes from its ancestor, PROOF. It inherits the heavy provisioning mechanism for the nodes by needing for each worker to be manual replicated, thus creating multiple points of failures and inconsistency. Furthermore, all the information about the client session environment is sand-boxed, forbidding external access to it.

1.3 MultiProc

Multiproc was introduced in 2015 allowing to process tasks in parallel on multicore processors. Unique in the landscape of scientific C++ libraries, this module leverages a multi-processing paradigm to express parallelism. Multiproc offers an interface inspired by the well known Python multiprocessing module but in addition provides reduce (merge) functionalities by implementing the deep-rooted map-reduce pattern.

Its goal was to address the problems of PROOF-Lite. For that, it was needed to exploit the fact that both client and workers would be spawned on the same machine and make it possible for the workers to access the client's memory. So MultiProc forks the original process multiple times to obtain several worker subprocesses whose memory is a copy of the original process, but it is not the same memory.

2 My mission - MultiProc module

Tree processing is a vital functionality for ROOT-based analysis and needs proper benchmarking tools. TProofBench is good example but it is only usable with PROOF. It is known that the packetizing algorithm, a part of the tree processing functionality, is a really simple one and could be more performant. Also the latest version of MultiProc module had some compiling errors and some bugs discovered during my work. Hence, my mission during this summer was to:

- Consolidate the existent version;
    - Complete the tree processing interface & bug fix;
- Develop a general purpose Benchmark tool;
- Run the benchmarking tests;
- Work on a new version of packetizing.

2.1 Tree processing interface

A TSelector object is used by the TTree::Draw, TTree::Scan, TTree::Process to navigate in a TTree and make selections.

The first step was to fix the TSelector processing interface that had some compilation errors. Once having solved the problems that TSelector had, I have enriched the TProcPool::ProcTree signatures. I added wrappers around the basic ProcTree method to support dataset definition as TFileCollection, TChain or a single file path, as you can see below.

```
TList* ProcTree(TTree& tree, TSelector& selector, …);
TList* ProcTree(const std::vector<std::string>& fileNames, TSelector& s
elector, …);
+ TList* ProcTree(const std::string &fileName, TSelector& selector, …);
+ TList* ProcTree(TFileCollection& files, TSelector& selector, …);
+ TList* ProcTree(TChain& files, TSelector& selector, …);
```

Having the tree processing interface working with TSelector was almost the end of this task. ROOT provides different types of documentation including a rich set of ROOT tutorials and code examples. Those tutorials are offered to developers to exercise specific functionality. So the last step of this task was to develop a new tutorial that uses a TSelector for tree processing.

2.2 Packetizing

The packetizing represents the way data is split in packets in order to be processed. The actual version that exists is working but is not suitable for our performance requests. Ideally the data should be split respecting the two following conditions:

1. The work needs to be split equally between workers;
2. The packets need to be as large as possible to minimize the impact of file open and init operations.

In this way we are sure that the workers are working continuously without waiting for more work.

2.2.1 V1 – Current version of packetizing

In order to understand why this version is unsatisfying, you need to understand how does it works. There are two cases that are treated differently.

First case

If there are more workers than files, for each file the entries are divided equally between workers. If you remember the two conditions listed above, in this case the second one is not respected.

Second case

If there are more files than workers, each worker processes one file at a time. When he finishes, the worker will get another file if there are still files to be processed or he will wait for the other workers to finish their job. Let's take an example: two workers and three files. Each worker gets one file. The one who finishes his work first, saying W1, will get the third file. As there is no more work to split, the other worker, W2 in this case, will just wait for the W1 to finish his work.

If we go back to the two conditions that need to be respected, in the example given above the first condition is not valid. The work is not split equally between the workers.

2.2.2 V1.2 – New version of packetizing

The big problem of the current version is how the work is split in the second case. This new version aims to fix that.

First case

We have more workers than files. In this case nothing changes.

Second case

We have more files than workers. Let's take the same example: two workers and three files. Each worker gets one file. In this version, the last file is split equally between workers. This solves the problem of latency and each worker gets the same amount of work.

* The benchmarking results presented below correspond to this version of packetizing.

2.3 Benchmarking Tool

The aim of this tool is to have the possibility to test the different technologies used for data processing in ROOT: Serial (TChain), PROOF-Lite and MultiProc.

The tool gives some flexibility to the user, letting him to set some parameters:

- The number of workers
- Local/remote ROOT files: the user is asked if the files to process are local or remote. Then the paths of files to be processed are read from a local file.
- With/without cache: this features permits to enable or disable the ROOT tree cache and the file system cache.
- The technology to be used: Serial or PROOF-Lite or MultiProc
- Path to the output file: the user needs to specify where the processing results should be stored.

2.4 Benchmarking – The first results

Having a first version of the benchmarking tool, we decided to run some tests. The technologies tested are PROOF-Lite and MultiProc. The tests were done on a 24 core Ubuntu machine. As we wanted to test the performance of the algorithms, we needed to avoid the influence of any equipment on the test runs. For that, the test files were read from the local hard disk. In this way the network did not influence our test results. The ROOT tree cache and the file system cache were also disabled, so that we can only have "cold reads".

Below I chose to represent two different cases showing the speed-up when processing four files and ten files, having a variable number of workers: one, two, four and eight.

First case

In the first graph four files were processed by a different number of workers. We see that the performance of PROOF-Lite and MultiProc are slightly the same for one, two and four workers. MultiProc tends to be better for eight workers. Remember that for these tests we are using the new version of packetizing. If the number of files is a multiple of number of workers, this means that workers are getting complete files to process. If not, the files are each one divided equally between workers. In the eight workers case each file is divided equally between workers.

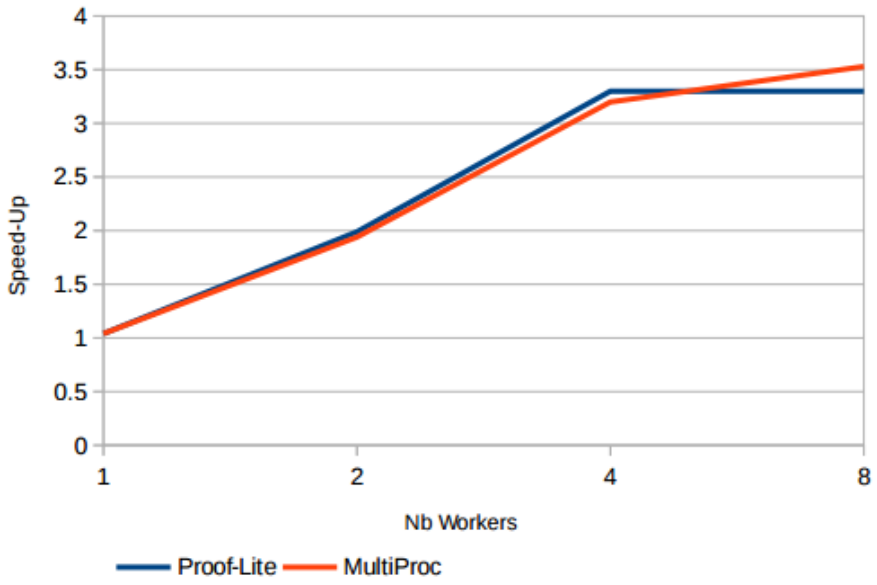

Figure 1 – MultiProc Speed-up - four files processed

Second case

In this case we have less workers than files. Let's take the eight workers example. As we have ten files, each worker gets one file to process and the last two files are each one split equally between workers. The performance is lower in this case because splitting the last two files in eight packets each one seems to be taking more time to process them and merge the results.

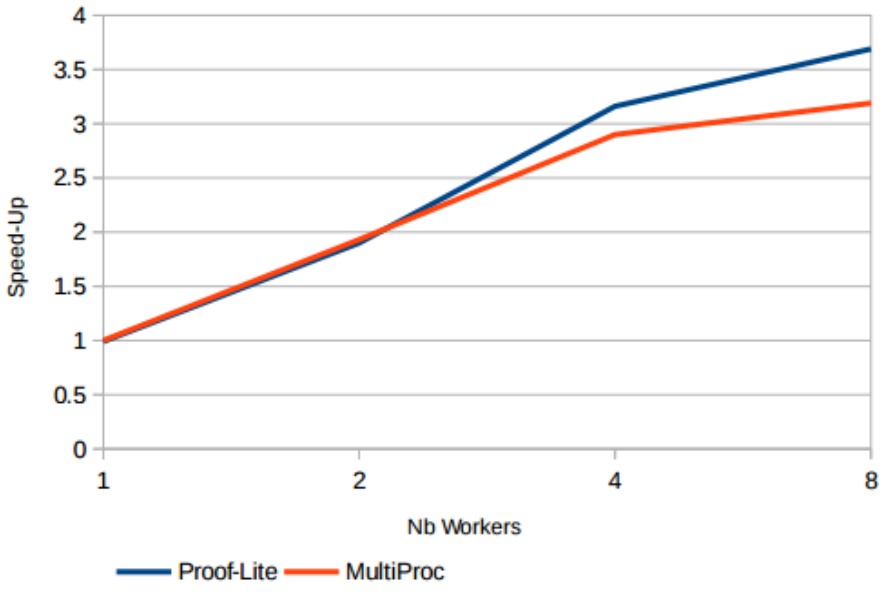

Figure 2 – MultiProc Speed-up - ten files processed

We conclude that in order to have a performant tool we need to strictly respect the two conditions of packetizing explained above. Therefore, we are looking for a new algorithm of packetizing.

2.5 Packetizing V2 – Future version of packetizing

Taking into account the problems of the actual version and respecting the two conditions, ideally we want to have this:

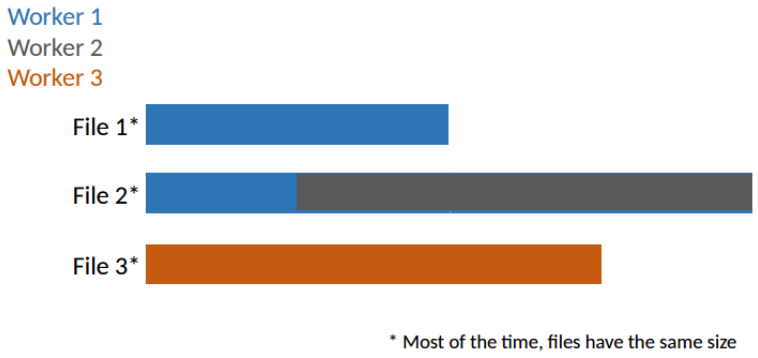

Figure 3 – Future version of packetizing

You can see in the example above that the work is split equally between workers and the packages are as large as possible. In order to be able to do this, we need to know the number of entries to be processed per file. This information is available in the header of each file. In order to get it, we need to open each file and read it. The problem is that the ROOT files can be very large and this would costs us a lot in terms of performance.

The information that we can easily have without opening the files is the file size. We already know the number of workers. With these two information we can make it work. Let's go back to the example given in the picture 1 and try to understand how this works.

In the example we have three workers and three files of different sizes:

- File 1: 1 GB
- File 2: 2 GB
- File 3:1,5 GB

In order to split the work equally, each worker needs to process 1/(number of workers), meaning 33% of the total work. Having the size of each file we know that 33% correspond to 1,5 GB. Therefore we are able to tell each worker the percentage of each file that he needs to process, as showed in the picture below:

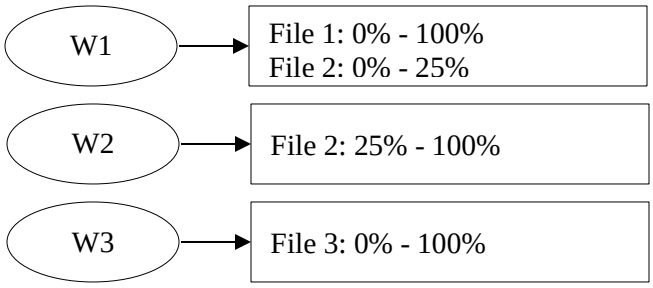

Figure 4 - The new version of packetizing

3 Future work

Concerning the future work, the new algorithm of packetizing needs to be implemented. Once developed, there should be run a new series of benchmarking tests in order to see the possible improvements on the packetizing technology. I know that a multi-threading module is also under development. Once finished, this new module should be benchmarked and compared to PROOF-Lite and MultiProc.

There are some improvements that should be done on the benchmarking tool. For instance, a visualization tool allowing you to read the bench results from a file and plotting a graph showing the speed-up would be welcomed.

4 Conclusion

Tree processing is a vital functionality. That's why TSelector was developed: it allows the user to navigate in a TTree, make selections and it is easy to use. MultiProc proposes now a TSelector processing interface ready to be used. A new packetizing version is now available permitting to speed-up the tree processing process in some cases. Also a new tool for benchmarking has been developed allowing to test the different processing technologies. This is very useful when some performance tests are needed to be done in case of new changes for instance.

5 References

[1] ROOT Users Guide

https://root.cern.ch/drupal/content/users-guide#UG

[2] ROOT Reference Guide

https://root.cern.ch/drupal/content/reference-guide

[3] C++ reference guide

http://en.cppreference.com/